DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming - Zombi defense

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Aldasoro Agustina	86/13	agusaldasoro@gmail.com
Rey Maximiliano	37/13	rey.maximiliano@gmail.com
Tirabasso Ignacio	718/12	ignacio.tirabasso@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1. Ejercicio 1

a) Tabla de descriptores de la GDT.

Al momento de armar la GDT, acorde a lo dispuesto en el enunciado, dejamos las primeras siete entradas de la tabla de descriptores libres. Luego, a partir de la posición 8 dejamos los descriptores de segmento pedidos: código y datos nivel 0 y código y datos nivel 3.

Armamos los cuatro segmentos de la GDT, llamándolos:

```
[GDT_IDX_CODE_0] = (gdt_entry) ;
[GDT_IDX_CODE_3] = (gdt_entry) ;
[GDT_IDX_DATA_0] = (gdt_entry) ;
[GDT_IDX_DATA_3] = (gdt_entry) ;
```

A los cuatro les seteamos el mismo *límite*: **0x26EFF** y la misma *base* en **0**, de este modo los cuatro descriptores de segmento direccionan a los primeros *623MB* de memoria. El *segment type* varía depende el segmento: CODE_0: **0x0A** (code execute/read), CODE_3: **0x0F** (code execute/read, conforming, accesed), DATA_0 y DATA_3: **0x02** (data read/write). El *Descriptor type* va en todos para system, por lo tanto es **0**. El *Descriptor privilege level* coincide con el nombre del descriptor (**0** para CODE_0 y DATA_0; **3** para CODE_3 y DATA_3). El bit de *Present* va para todos en **1** y los bit de *Available for use by system software* y *l* van para todos en **0**. El bit de *Default operation size* va para todos en **1** porque es un código de 32bits. El bit de *Granularity* va para todos en **1**.

b) Pasaje a modo protegido y seteo de la pila del kernel.

Para pasar a modo protegido los pasos que debemos llevar a cabo son:

- ▷ Completar la GDT (resuelto en el inciso A).
- ▷ Deshabilitar interrupciones (para ello se ejecuta la instrucción *cli*).
- ▷ Habilitar A20 (en nuestro caso se resuelve haciendo *call habilitar_A20*).
- ▷ Cargar el registro GDTR con la dirección base de la GDT (lo hacemos con la instrucción *lgdt [GDT_DESC]* la etiqueta GDT_DESC apunta al descriptor de la GDT en el código).
- ▷ Una vez hecho esto, estamos en condiciones de setear el bit PE del registro CR0 (debemos hacer: *mov eax,cr0 ; or eax,1 ; mov cr0,eax*).
- ▷ Lo siguiente a realizar es el *jump far* a la siguiente instrucción (utilizamos el selector de segmento *0x50* y de offset la etiqueta *modo_protegido* por que el selector es 0x50??).
- ▷ Una vez ya en modo protegido, nos encontramos trabajando en 32 bits y ahora es cuando cargamos los registros de segmento (a los registros *es, ds, ss y gs* les asignamos el valor de *0x40* y al registro *fs* le asignamos el valor *0x60* aca habría que poner el porque de estos valores??).

Para setear la pila del kernel en la dirección 0x27000 debemos llevar a cabo la siguiente instrucción:

```
mov ebp, 0x27000
```

c) Segmento adicional que describe el área de la pantalla en memoria que puede ser utilizado sólo por el kernel.

HELP! aca no se que ponerrrrrrrr

```
; Cambiar modo de video a 80 X 50
mov ax, 0003h
int 10h ; set mode 03h
xor bx, bx
mov ax, 1112h
int 10h ; load 8x8 font
```

d) Rutina que se encarga de limpiar la pantalla y pintar el área del mapa

En este punto debemos establecer un fondo de color verde, junto con las dos barras laterales para cada uno de los jugadores (una roja y otra azul). Para esto, debemos contar con una función que limpie la pantalla en un primer momento: *clear_screen*.

La función ***clear_screen***, implementada en lenguaje C, se va a encargar de:

- ▷ Guardar en una variable local: *size* el tamaño de la pantalla (VIDEO_COLS * VIDEO_FILS).
- ▷ Luego va a hacer un while desde *0* hasta *size* que, empezando por la dirección donde está almacenada la memoria de video, vaya guardando el caracter que es todo negro.

De este modo, logramos hacer que toda el área de la pantalla quede “pintada” de negro.

En segunda instancia, armamos la función *print_map*. La función ***print_map***, también implementada en lenguaje C, con el fin de pintar el área del mapa con los colores deseados, posee el siguiente comportamiento:

- ▷ En un primer momento, llama a la función *clear_screen*.
- ▷ Se arman cuatro variables locales: blue, red, green y black. Cada una de ellas es un caracter completamente de su color.
- ▷ Mediante dos *fors* anidados recorremos toda el área del mapa, y dependiendo de la posición en la que se encuentre es el color que le va a ser asignado (rojo a las primeras dos columnas de la izquierda, azul a las últimas dos, negro a las últimas cinco filas y el resto en verde).

Luego este código fue complejizado para que pueda imprimir los puntos de los jugadores y sus respectivos puntajes por zombie tal cual debe ser cuando comienza el juego.

2. Ejercicio 2

ESTO HAY QUE HACERLO!!!!!!!!

3. Ejercicio 3

Limpiar el buffer de video es hacer clear_screen???... Asumo que sí.

a) Implementación completa de print_map.

En el ejercicio 1 ya habíamos logrado armar **clear_screen** y **print_map**, esta última contaba con una funcionalidad acotada. Sólo armaba cuatro bloques de colores. Ahora lo que vamos a hacer es extender la función **print_map** para que escriba los puntos de los jugadores y marque el estado de los zombies, tal como aparece en la *figura 9* del enunciado.

La función print_map la vamos a extender sumándole las siguientes instrucciones:

- Debemos armar los *cuadrados de puntaje* para el jugador rojo y para el jugador azul. Nos movemos entre las últimas cinco filas, desde la columna 35 hasta la 39 se pintan con el caracter completamente rojo y desde la columna 40 hasta la 44 se pintan con el caracter completamente azul.
- Escribimos en el centro de cada cuadrado el puntaje inicial: 0. Para ello creamos dos caracteres que sean 0 y cada uno con los atributos necesarios para que sean: fondo rojo, letra blanca y fondo azul, letra blanca.
- Luego escribimos la cantidad de zombies restantes, para lo cual se utilizan dos caracteres idénticos a los mencionados arriba. Estos puntajes se ubican cada uno a un costado de su cuadrado respectivo.
- Por último, resta escribir el estado de los zombies. Generamos un char* que sean todos los números de zombies y dándole formato de fondo negro, caracter blanco los copiamos dos veces: uno para el jugador azul y otro para el jugador rojo. Como todos los zombies se encuentran disponibles, generamos otro char* que sean diez 'x' y ubicamos en su posición correspondiente uno que posea los atributos de fondo negro, caracter azul y otro de fondo negro, caracter rojo.

Con el objetivo de hacer que lo descripto anteriormente sea una tarea más simple contamos con las funciones: *print_string* y *get_format*.

La función **print_string** recibe una posición en el mapa (x,y), un char* con el texto que queremos imprimir en pantalla y un short con los atributos deseados. PONER ALGO DE AVOID PRINT BUG. Dentro de un *for* que recorre horizontalmente la posición en memoria a partir de la posición (x,y), se va avanzando el char* y en cada iteración se le asigna a esa posición de memoria el char actual con los atributos pasados por parámetro.

La función **get_format** es una función simple la cual recibe como parámetro los atributos deseados y los devuelve con el formato de un sólo char, que es el que debemos utilizar.

```
unsigned char getFormat(unsigned char fore_color, char fore_bright, unsigned char back_color,
                        char blink) {
    return fore_color | fore_bright | back_color | blink;
}
```

HASTA ACA HICE, DE ACA PARA ABAJO NO DOY FE DE NADA...

b) Rutinas encargadas de inicializar el directorio y tablas de páginas para el kernel.

Se muestran las rutinas encargadas de inicializar el directorio y tablas de páginas para el kernel (mmu_inicializar_dir_kernel).

```
void mmu_inicializar_dir_kernel() {
    page_directory *pd = (page_directory *) 0x27000;
    int i;

    /* Creo 1024 entradas en page_directory con todo cero. */
    for (i = 0; i < 1024; i++) {
        pd[i] = (page_directory) {};
    }

    /* Mapeo las primeras 4 (0,1,2,3) entradas del page directory
    con la base y permisos correspondientes. */
    for(i = 0; i < 4; i++) {
        pd[i] = (page_directory) {
            .base = 0x28 + i,
            .rw = 0x1,
            .p = 1,
        };
    }

    page_table* pt = (page_table*) 0x28000;
    for(i = 0; i < 1024; i++) {
        pt[i] = (page_table) {
            .base = i,
            .rw = 0x1,
            .p = 1,
        };
    }
}
```

c) Completamos el código necesario para activar paginación.

```
    ; Habilitar paginacion

    mov eax,0x27000
    mov cr3,eax

    mov eax,cr0
    or eax,0x80000000
    mov cr0,eax


    mov eax,0x100000
    mov cr3,eax
```

4. Ejercicio 4

a) Para administrar la memoria en el área libre, tenemos un contador de páginas utilizadas denominándolo *páginas*. Luego contamos con las funciones *get_page_directory* y *get_page_table* las cuales nos brindan un nuevo page directory o una nueva page table correspondientemente.

```
page_directory* get_page_directory() {
    page_directory* pd = (page_directory*) PAGES;
    pd += paginas * 4096;
    int i;
    for (i = 0; i < 1024; i++) {
        pd[i] = (page_directory) {};
    }

    pd[0] = (page_directory) {
        .base = 0x28,
        .rw = 0x1,
        .p = 1
    };

    paginas++;

    return pd;
}
```

```
page_table* get_page_table() {
    page_table* pgt = (page_table*) PAGES;
    pgt += paginas * 0x1000;
    int i;
    for (i = 0; i < 1024; i++) {
        pgt[i] = (page_table) {};
    }

    paginas++;

    return pgt
    };
}
```

b) La rutina *mmu_inicializar_dir_zombi* se encarga de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 6. Copia el código de la tarea a su área asignada, es decir la posición indicada por el jugador dentro del mapa y mapea dichas páginas a partir de la dirección virtual 0x08000000(128MB).

```
/* Guerrero = 0, Mago = 1, Clerigo = 2
   player = 0 es A
   player = 1 es B
*/
  page_directory* mmu_inicializar_dir_zombie(unsigned int player, unsigned char class,
  unsigned int y) {

    page_directory* pd = get_page_directory();
    unsigned int x = (player ? 79 : 2);

    unsigned int offset_x[9] = {0, -1, -1, -1, 0, 0, 1, 1, 1};
    unsigned int offset_y[9] = {0, 0, -1, 1, -1, 1, 0, 1, -1};

    /* En este for se mapean las nueve páginas correspondientes a un jugador */
    int i, _x, _y;
    for(i = 0; i < 9; i++) {
        _x = y + offset_x[i] * (player ? 1 : -1);  \\ x es la posición x dentro del mapa
        _y = x + offset_y[i] * (player ? 1 : -1);  \\ y es la posición y dentro del mapa

        /* La función get_physical_address devuelve la dirección física en memoria
        a la cual le corresponde el par (x, y) pasado por parámetro */
        mmu_mapear_pagina(0x8000000 + (i*0x1000), pd, get_physical_address(_x, _y), 1, 0);
    }

    /* Código para copiar la tarea del Zombie */
    int address = address = 0x10000 + (player ? 0 : 1)  * 0x3000 + class * 0x1000;
    i = 0;
    unsigned char *code = (unsigned char *) 0x8000000;
    unsigned char * paddress = (unsigned char*) address;
    while (i++ < 0x1000) {
        code[i] = paddress[i];
    }

    return pd;
}
```

c) La rutina *mmu_mapear_página* permite mapear la página física correspondiente a su física en la dirección virtual utilizando cr3.

```
void mmu_mapear_pagina(unsigned int virtual, page_directory* pd, unsigned int fisica,
   unsigned char rw, unsigned char us) {

    unsigned int directory = (virtual >> 22);
    unsigned int table     = (virtual & 0x003FF000) >> 12;

    page_table* pt = (page_table*) (pd[directory].base << 12);

    if (pd[directory].p == 0){
        pd[directory].base = ((unsigned int) get_page_table()) >> 12;
        pd[directory].rw = rw;
    pd[directory].us = us;
        pd[directory].p = 1;
    }

    pt = (page_table*) (pd[directory].base << 12);
    pt[table].base = fisica >> 12;
    pt[table].rw = rw;
    pt[table].us = us;
    pt[table].p = 1;

    tlbflush();
}
```

mmu_unmapear_pagina borra el mapeo creado en la dirección virtual virtual utilizando cr3.

```
 void mmu_unmapear_pagina(unsigned int virtual,page_directory* cr3){
  unsigned int directory = (virtual >> 22);
    unsigned int table     = (virtual & 0x003FF000) >> 12;

    page_table* pt = (page_table*) (cr3[directory].base << 12);

    if (cr3[directory].p != 0){
        pt = (page_table*) (cr3[directory].base << 12);
        pt[table].p = 0;
    }

    tlbflush();
 }
```

5. Ejercicio 5

a) Completamos las entradas necesarias en la IDT para asociar una rutina a la interrupción del reloj, otra a la interrupción de teclado y por último una a la interrupción de software 0x66. Es decir las posiciones 32,33 y 66.

b) A continuación, la rutina asociada a la interrupción del reloj, para que por cada tick llame a la función screen próximo reloj. La misma se encarga de mostrar cada vez que se llame, la animación de un cursor rotando en la esquina inferior derecha de la pantalla.

```
;; Rutina de atención del RELOJ

global _isr32
_isr32:
    pushad
    call proximo_reloj   ; Ya definida en isr.asm
    call proximo_indice  ;Devuelve el próximo índice en la GDT a ejecutar

    cmp ax,0
    je .nojump

    mov [sched_tarea_selector], ax
    call fin_intr_pic1
    jmp far [sched_tarea_offset]
    jmp .end

.nojump:
    call fin_intr_pic1

.end:
    ; switchear tareas.
    popad
    iret
```

c) Ahora, la rutina asociada a la interrupción de teclado de forma que si se presiona cualquiera de las teclas a utilizar en el juego, se presenta la misma en la esquina superior derecha de la pantalla.

```
;; Rutina de atención del TECLADO

global _isr33
extern printf
extern print_int
extern handle_keyboard_interrumption
_isr33:
    pushad
    xor eax,eax
    in al, 0x60

    mov dword [esp], eax
    call handle_keyboard_interrumption

    mov dword  [esp + 0x], 0
    mov dword [esp + 0xc], 67
    mov dword [esp + 0x8], keyboard_str
    mov dword [esp + 0x4], eax
    call printf    ;función implementada por nosotros

    call fin_intr_pic1
    popad
    iret
```

d) Escribimos la rutina asociada a la interrupción 0x66 para que modifique el valor de eax por 0x42.

```
;; Rutina de atención 0x66

global _isr66
_isr33:

    mov eax,0x42

    iret
```

6. Ejercicio 6

a) Definimos tres entradas en la GDT que consideramos necesarias para ser usadas como descriptores de TSS: una para ser utilizada por la tarea inicial, otra para la tarea actual y una última para la tarea siguiente.

b) Completamos la entrada de la TSS de la tarea Idle con la información de la tarea Idle. La tarea Idle se encuentra en la dirección 0x00016000. La pila se alojará en la misma dirección que la pila del kernel y será mapeada con identity mapping. Esta tarea ocupa 1 pagina de 4KB y debe ser mapeada con identity mapping. Además la misma comparte el mismo CR3 que el kernel.

```
void tss_inicializar() {
    int i = 0;
    while(i < CANT_ZOMBIS) {
        inUseA[i] = 0;
        inUseB[i] = 0;
        i++;
    }
    currentZombieA = 0;
    currentZombieB = 0;

    // inicializar tss_idle
    tss_inicializar_tarea_idle();

    memcpy(&tss_idle, &tss_inicial, sizeof(tss));
    memcpy(&tss_idle, &current_task, sizeof(tss));
    memcpy(&tss_idle, &next_task, sizeof(tss));

    gdt[GDT_INITIAL_TSS].base_31_24 = ((u32) (&tss_inicial) & 0xFF000000) >> 24;
    gdt[GDT_INITIAL_TSS].base_23_16 = ((u32) (&tss_inicial) & 0x00FF0000) >> 16;
    gdt[GDT_INITIAL_TSS].base_0_15  = (u32) (&tss_inicial) & 0x0000FFFF;

    gdt[GDT_CURRENT_TSS].base_31_24 = ((u32) (&current_task) & 0xFF000000) >> 24;
    gdt[GDT_CURRENT_TSS].base_23_16 = ((u32) (&current_task) & 0x00FF0000) >> 16;
    gdt[GDT_CURRENT_TSS].base_0_15  = (u32) (&current_task) & 0x0000FFFF;
}
```

```
void tss_inicializar_tarea_idle() {

    tss_idle = (tss) {};

    tss_idle.eip = 0x00016000;
    tss_idle.cr3 = 0x27000;

    tss_idle.ebp = 0x27000;
    tss_idle.esp = 0x27000;

    tss_idle.es = 0x40;
    tss_idle.ds = 0x40;
    tss_idle.ss = 0x40;
    tss_idle.gs = 0x40;
    tss_idle.cs = 0x50;

    tss_idle.eflags = 0x202;
    tss_idle.iomap = 0xffff;
}
```

d) Código necesario para ejecutar la tarea Idle, es decir, saltar intercambiando las TSS, entre la tarea inicial y la tarea Idle:

```
idle:
    .loopear:
        inc dword [numero]
        cmp dword [numero], 0x4
        jb .imprimir

    .reset_contador:
        mov dword [numero], 0x0

    .imprimir:
        ; Imprimir 'reloj'
        mov ebx, dword [numero]
        add ebx, message1
        imprimir_texto_mp ebx, 1, 0x0f, 49, 76
        mov ebx, chirimbolo_open
        imprimir_texto_mp ebx, 1, 0x0f, 49, 76-1
        mov ebx, chirimbolo_close
        imprimir_texto_mp ebx, 1, 0x0f, 49, 76+1

    jmp .loopear
```

7. Ejercicio 7

c) Se muestra la rutina de la interrupción 0x66, para que implemente el servicio mover según se indica en la sección 3.1.1.

```
global _isr66
extern movimiento
_isr66:
    pushad

    push dx
    push esi
    push edi
    push eax
    call movimiento

    popad
    iret
```